Algorithms for Dense Matrices II

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 15/16

Topics

Data parallel algorithms for dense matrices

- Matrix-Vector Multiplication
- Matrix-Matrix Multiplication

Matrix-Vector Multiplication

Compute $y = Ax$, matrix $A \in \mathcal{R}^{N \times M}$ and vector $x \in \mathcal{R}^M$

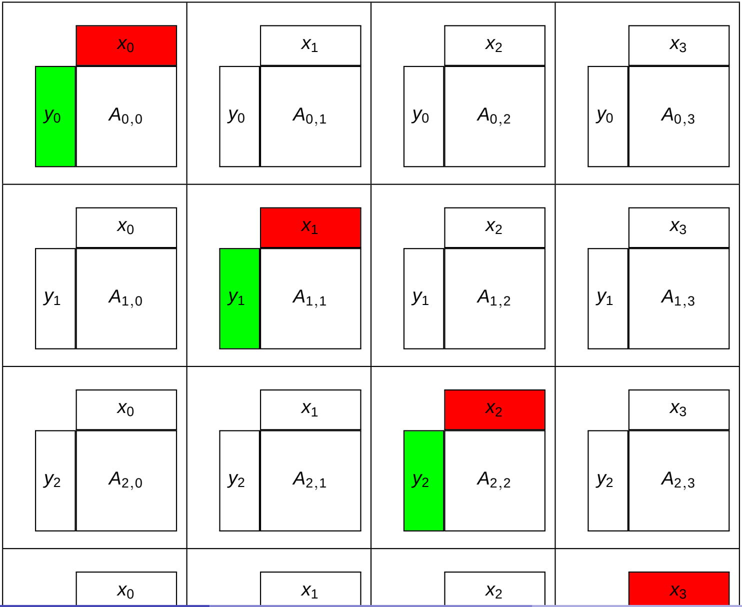
- Different possibilities for data partitioning
- Distribution of the matrix and the vector have to fit together
- Distribution of the result vector $y \in \mathcal{R}^N$ same as of input vector x

Example:

- Matrix is blockwise distributed onto an array topology
- Input vector x is correspondingly distributed blockwise across the diagonal processors
- The processor array is quadratic
- Vector segment x_q is needed in each processor column and is therefore to copy in each column (one-to-all).
- Local computation of the product $y_{p,q} = A_{p,q} x_q$.
- Complete segment y_p results first from the summation $y_p = \sum_q y_{p,q}$. (further all-to-one communication)
- Result can immediately used for further matrix-vector multiplications

Matrix-Vector Multiplication: Partitioning

Partitioning for the Matrix-Vector product



Matrix-Vector Multiplication: Parallel Runtime

Parallel runtime for a $N \times N$ matrix and $\sqrt{P} \times \sqrt{P}$ processors with cut-through communication networks:

$$
\begin{aligned}
T_P(N,P) &= \underbrace{\left(t_s + t_h + t_w \overbrace{\frac{N}{\sqrt{P}}}^{\text{vector}}\right) \operatorname{ld}\sqrt{P}}_{\substack{\text{Distribute } x \\ \text{across column}}} + \underbrace{\left(\frac{N}{\sqrt{P}}\right)^2 2t_f}_{\substack{\text{local matrix-vector} \\ \text{mult.}}} \\
&\quad + \underbrace{\left(t_s + t_h + t_w \frac{N}{\sqrt{P}}\right) \operatorname{ld}\sqrt{P}}_{\substack{\text{reduction} \\ (t_f \ll t_w)}} = \\
&= \operatorname{ld}\sqrt{P}(t_s + t_h)2 + \frac{N}{\sqrt{P}} \operatorname{ld}\sqrt{P}2t_w + \frac{N^2}{P}2t_f
\end{aligned}
$$

For fixed P and $N \to \infty$ the communication share get arbitrary small, thus an iso-efficiency function exists, the algorithm is scalable.

Matrix-Vector Multiplication: Work/Overhead

Let us compute work and overhead:

Recalculate to the work W:

$$
\begin{aligned}
W &= N^2 2t_f \text{ (seq. runtime)} \\
&\Rightarrow \quad N = \frac{\sqrt{W}}{\sqrt{2t_f}} \\
T_P(W,P) &= \operatorname{ld}\sqrt{P}(t_s + t_h)2 + \frac{\sqrt{W}}{\sqrt{P}} \operatorname{ld}\sqrt{P} \frac{2t_w}{\sqrt{2t_f}} + \frac{W}{P}
\end{aligned}
$$

Overhead:

$$
T_O(W,P) = PT_P(W,P) - W =
$$

$$
= \sqrt{W}\sqrt{P} \operatorname{ld}\sqrt{P} \frac{2t_w}{\sqrt{2t_f}} + P \operatorname{ld}\sqrt{P}(t_s + t_h)2
$$

Matrix-Vector Multiplication: Iso-Efficency

and now the iso-efficiency function:

Iso-efficiency $(T_O(W,P) \overset{!}{=} KW)$: T_O has two terms.

For the first we achieve

$$
\begin{aligned}
& \sqrt{W}\sqrt{P}\,\mathrm{ld}\,\sqrt{P}\,\frac{2t_w}{\sqrt{2t_f}} = KW \\
\iff \quad & W = P(\mathrm{ld}\,\sqrt{P})^2\,\frac{4t_w^2}{2t_f K^2}
\end{aligned}
$$

and for the second

$$
\begin{aligned}
& P\,\mathrm{ld}\,\sqrt{P}(t_s + t_h)2 = KW \\
\iff \quad & W = P\,\mathrm{ld}\,\sqrt{P}\,\frac{(t_s + t_h)2}{K};
\end{aligned}
$$

thus $W = 0(P(\mathrm{ld}\,\sqrt{P})^2)$ is the desired iso-efficiency function.

Matrix-Matrix Multiplication

Algorithm of Cannon

It is to calculate $C = A \cdot B$.

- The $N \times N$ matrices A and B, that are to multiply, are blockwise distributed onto a 2D-array topology ($\sqrt{P} \times \sqrt{P}$)
- For practical reasons should be the result C again be stored in the same partitioning.
- Process (p, q) has thus

$$C_{p,q} = \sum_k A_{p,k} \cdot B_{k,q}$$

to calculate, needs therefore block row p of A and block column q of B.

Matrix-Matrix Multiplication

The two phases of Cannon's algorithm are

1. *Alignment phase:* The blocks of A are shifted in each row cyclic to the left, until the diagonal blocks reside in the first column. Corresponding one shifts the blocks of B in the columns to above, until all diagonal blocks reside in the first row.
   After the alignment phase processor (p, q) has the blocks

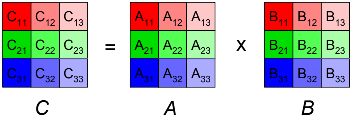
$$A_{p,\underbrace{(q+p)\,\%\,\sqrt{P}}} \qquad \text{(row } p \text{ shifts } p \text{ times to the left)}$$

$$B_{\underbrace{(p+q)\,\%\,\sqrt{P}},q} \qquad \text{(column } q \text{ shifts } q \text{ time to above).}$$

2. *Computing phase:* Obviously now each process stores two fitting blocks, that it can multiply. Are the blocks of A in each row of A shifted for one position to the left and the one of B in each column to the above, then each owns again two fitting blocks. After $\sqrt{P}$ steps the result is ready.

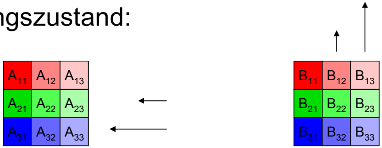
Cannon's Algorithm

- Is based on blockwise partitioning of the matrices
- Setup phase
  - Rotation of the matrizen A and B
- Iteration over $\sqrt{p}$ steps
  - Compute locally block matrix product
  - Shift A horizontally and B vertically



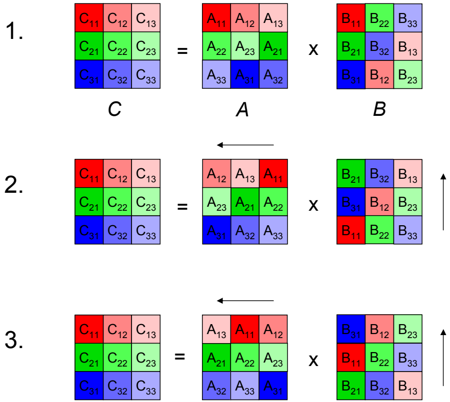
Cannon's Algorithm - Rotation

- Anfangszustand:



- Verdrehen: Rotieren der i. Zeile (Spalte) von A (B) um i-Schritte:

Cannon's Algorithm - Iteration



1. $C_{11} = A_{11}B_{11}$

2. $C_{11} = A_{11}B_{11} + A_{12}B_{21}$

3. $C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$

Cannon's Algorithm - Performance Analysis

- A, B verdrehen: $2 * s * (t_{startup} + t_{word} N^2/p)$
- Iteration (s-mal):
  - dgemm: $2 * t_{flop} * (n/s)^3 = 2 * t_{flop} * n^3/p^{1,5}$
  - A, B rollen: $2 * (t_{startup} + t_{word} N^2/p)$
- Gesamt: $t_{cannon}(p) = 4t_{startup} * s + 4t_{word} * N^2/s + 2t_{flop} * N^3/p$

- Effizienz $= 2\, t_{flop} * N^3 / (p * t_{cannon}(p))$
  $= 1 / (2t_{startup} * (s/N)^3 + 2t_{word} * s/N + t_{flop})$
  $\approx 1 / O(1 + sqrt(p) / N))$
- Effizienz $\rightarrow$ 1, wenn $(N/s) \rightarrow \infty$
  - N / s = N / sqrt(p) = sqrt(Daten pro Prozessor)

Cannon with MPI (Init)

```
/* Baue Gitter und hole Koordinaten */
int   dims[2], periods[2] = {1, 1};
int   mycoords[2];

dims[0] = sqrt(num_procs);
dims[1] = num_procs / dims[0];

MPI_Cart_create(MPI_COMM_WORLD,  /* kollektiv */
                2, dims, periods,
                0, &comm_2d);
MPI_Comm_rank(comm_2d, &my2drank);
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

/* Lokale Blöcke der Matrizen */
double      *a, *b, *c;

/* Lade a, b  und c entsprechend der Koordinaten */
 ...
```

Cannon with MPI (Rotate)

```
/* Matrix-Verdrehung A */
MPI_Cart_shift(comm_2d, 0, -mycoords[0],
               &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
                     shiftdest, 77, shiftsource, 77,
                     comm_2d, &status);

/* Matrix-Verdrehung B */
MPI_Cart_shift(comm_2d, 1, -mycoords[1],
               &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
                     shiftdest, 77, shiftsource, 77,
                     comm_2d, &status);
```

Cannon with MPI (Iteration)

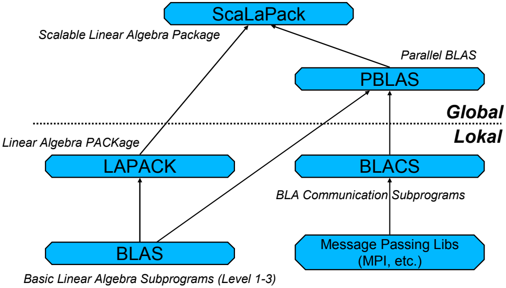
```
/* Finde linken und oberen Nachbarn */
MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

for (i=0; i<dims[0]; i++)
{
 dgemm(nlocal, a, b, c); /* c= c + a * b */

 /* Matrix A nach links rollen */
 MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
                      leftrank, 77, rightrank, 77,
                      comm_2d, &status);

 /* Matrix B nach oben rollen */
 MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
                      uprank, 77, downrank, 77,
                      comm_2d, &status);
}

/* A und B zurück in Ursprungs-Zustand */
 …
```

Cannon - Practical Aspects

- Efficient, but not simple generalisation, if
  - Matrices are not quadratic
  - Dimensions are not without rest divisible by p
  - Other matrix partitionings are needed
- Iso-efficiency function of Cannon's Algorithmus: $O(P^{3/2})$, $N/\sqrt{P} = const \rightarrow$ Efficiency remains constant for fixed block sizes per processor and increasing processor count
- Dekel-Nassimi-Salmi algorithm enables the usage of N^3 processors (Cannon N^2) with better iso-efficiency function.

Standard Libraries for Linear Algebra

ATLAS, BLITZ (expression templates), ISTL (generic programming), ScaLaPack (classical package), Trilinos (huge code family), `http://www.netlib.org`



Matrix-Matrix Multiplication: Iso-efficiency Analysis

Consider the corresponding iso-efficiency function.

Sequential runtime :

$$W = T_S(N) = N^3 2t_f$$
$$\Rightarrow N = \left(\frac{W}{2t_f}\right)^{\frac{1}{3}}$$

Parallel runtime:

$$\begin{aligned}
T_P(N,P) &= \underbrace{\left(\sqrt{P}-1\right)\left(t_s + t_h + t_w \frac{N^2}{P}\right)}_{\text{alignment}} \overbrace{4}^{\textbf{send/recv } A/B} \\
&\quad + \sqrt{P}\left(\underbrace{\left(\frac{N}{\sqrt{P}}\right)^3 2t_f}_{\text{multiplic. of a block}} + \left(t_s + t_h + t_w \frac{N^2}{P}\right) 4 \right) \approx \\
&\approx \sqrt{P}(t_s + t_h)8 + \frac{N^2}{\sqrt{P}} t_w 8 + \frac{N^3}{P} 2t_f \\
T_P(W,P) &= \sqrt{P}(t_s + t_h)8 + \frac{W^{\frac{2}{3}}}{\sqrt{P}} \frac{8t_w}{(2t_f)^{\frac{1}{3}}} + \frac{W}{P}
\end{aligned}$$

Matrix-Matrix Multiplication: Iso-efficiency Analysis

Overhead:

$$T_O(W,P) = PT_P(W,P) - W = P^{\frac{3}{2}}(t_s + t_h)8 + \sqrt{P}W^{\frac{2}{3}}\frac{8t_w}{(2t_f)^{\frac{1}{3}}}$$

Result:

- Thus is $W = O(P^{3/2})$.
- Because of $N = \left(\frac{W}{2t_f}\right)^{1/3}$ applies $N/\sqrt{P} = const$
- Thus for *fixed* block size in each processor and increasing processor count the efficiency keeps constant.
- If we restrict in the algorithm of Cannon to 1×1 blocks per processor, thus $\sqrt{P} = N$, then we can use for the required N^3 multiplications only N^2 processors.
- This is the reasion for the iso-efficiency function of order $P^{3/2}$.

Matrix-Matrix Multiplication: Dekel-Nassimi-Salmi-Alg.

Dekel-Nassimi-Salmi-Algorithm

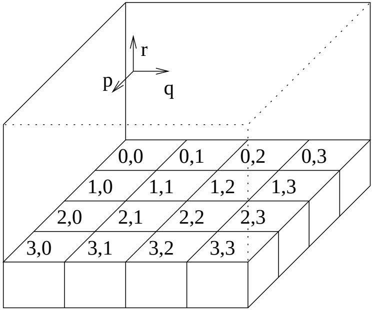
- Now we consider an algorithm that renders the usage of up to N^3 processors for a $N \times N$ matrix possible.
- Given are then $N \times N$ matrices A and B as well as a 3D array of processors of dimension $P^{1/3} \times P^{1/3} \times P^{1/3}$.
- The processors are addressed by the coordinates (p, q, r).
- To calculate the block $C_{p,q}$ of the result matrix C via

$$C_{p,q} = \sum_{r=0}^{P^{\frac{1}{3}}-1} A_{p,r} \cdot B_{r,q} \qquad (1)$$

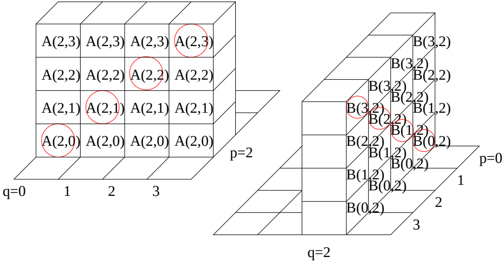
  we use $P^{1/3}$ processors, in detail processor (p, q, r) is exactly responsible for the product $A_{p,r} \cdot B_{r,q}$.
- Now is still to decide, how the input and result matrices shall be distributed.
- Both A and B are partitioned into $P^{1/3} \times P^{1/3}$ blocks of size $\frac{N}{P^{1/3}} \times \frac{N}{P^{1/3}}$.
- $A_{p,q}$ and $B_{p,q}$ is stored in the beginning in processor $(p, q, 0)$, also the result $C_{p,q}$ shall reside there.
- The processors (p, q, r) for $r > 0$ are only used temporarily.

Matrix-Matrix Multiplication: Dekel-Nassimi-Salmi Alg.

Distribution of A, B, C for $P^{1/3} = 4$ (P=64).



Partitioning of the blocks of A and B (at the beginning) and C (at the end)



Distribution of A and B for the multiplication

Matrix-Matrix Multiplication: Dekel-Nassimi-Salmi Alg.

- That now each processor (p, q, r) can perform „its" multiplication $A_{p,r} \cdot B_{r,q}$, the involved blocks of A and B first have to be moved to the right position.
- All processors require $(p, *, r)$ the block $A_{p,r}$ and all processors $(*, q, r)$ the block $B_{r,q}$.
- The distribution is achieved in the following way:

  Processor $(p, q, 0)$ sends $A_{p,q}$ to processor (p, q, q) and sends then (p, q, q) the $A_{p,q}$ to all $(p, *, q)$ via a one-to-all communication on $P^{1/3}$ processors. Corresponding sends $(p, q, 0)$ the $B_{p,q}$ to processor (p, q, p), and this distributed then to $(*, q, p)$.

- After the multiplication in each (p, q, r) the results of all $(p, q, *)$ are still to collect in $(p, q, 0)$ via a all-to-one communication on $P^{1/3}$ processors.

Matrix-Matrix Multiplication: Dekel-Nassimi-Salmi Alg.

Let us analyse the method in detail (3D-cut-through network):

$$
\begin{aligned}
W &= T_S(N) = N^3 2t_f \quad \Rightarrow N = \left(\frac{W}{2t_f}\right)^{\frac{1}{3}} \\
T_P(N,P) &= \underbrace{\left(t_s + t_h + t_w\left(\frac{N}{P^{\frac{1}{3}}}\right)^2\right)}_{(p,q,0)\longrightarrow(p,q,q),(p,q,p)} \overbrace{2}^{\substack{A_{p,q}\text{ u.}\\ B_{p,q}}} + \underbrace{\left(t_s + t_h + t_w\left(\frac{N}{P^{\frac{1}{3}}}\right)^2\right) \operatorname{ld} P^{\frac{1}{3}} \overbrace{2}^{A,B}}_{\text{one-to-all}} \\
&+ \underbrace{\left(\frac{N}{P^{\frac{1}{3}}}\right)^3 2t_f}_{\text{multiplication}} + \underbrace{\left(t_s + t_h + t_w\left(\frac{N}{P^{\frac{1}{3}}}\right)^2\right) \operatorname{ld} P^{\frac{1}{3}}}_{\text{all-to-one } (t_f \ll t_w)} \approx \\
&\approx 3 \operatorname{ld} P^{\frac{1}{3}} (t_s + t_h) + \frac{N^2}{P^{\frac{2}{3}}} 3 \operatorname{ld} P^{\frac{1}{3}} t_w + \frac{N^3}{P} 2t_f \\
T_P(W,P) &= 3 \operatorname{ld} P^{\frac{1}{3}} (t_s + t_h) + \frac{W^{\frac{2}{3}}}{P^{\frac{2}{3}}} 3 \operatorname{ld} P^{\frac{1}{3}} \frac{t_w}{(2t_f)^{\frac{2}{3}}} + \frac{W}{P} \\
T_O(W,P) &= P \operatorname{ld} P^{\frac{1}{3}} 3(t_s + t_h) + W^{\frac{2}{3}} P^{\frac{1}{3}} \operatorname{ld} P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}}}
\end{aligned}
$$

Matrix-Matrix Multiplication: Dekel-Nassimi-Salmi Alg.

- From the second term of $T_O(W, P)$ we approximate the iso-efficiency function:

$$W^{\frac{2}{3}} P^{\frac{1}{3}} \operatorname{ld} P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}}} = KW$$

$$\iff \quad W^{\frac{1}{3}} = P^{\frac{1}{3}} \operatorname{ld} P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}} K}$$

$$\iff \quad \boxed{W = P \left(\operatorname{ld} P^{\frac{1}{3}} \right)^3 \frac{27 {t_w}^3}{4 {t_f}^2 K^3}.}$$

- Thus we achieve the iso-efficiency function $O(P(\operatorname{ld} P^{\frac{1}{3}})^3)$ and therefore a better scalability than for Cannon's algorithm.
- We have always assumed, that the optimal sequential complexity of the matrix multiplication is N^3. The algorithm of Strassen has however a complexity of $O(N^{2.87})$.
- For an efficient implementation of the multiplication of two matrix blocks on a processor one has to ensure cache efficiency.